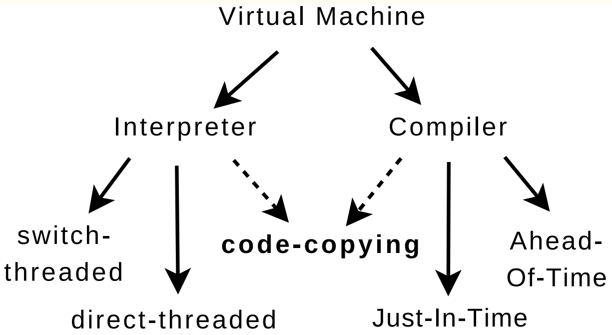
Compiler-guaranteed Safety in Code-copying Virtual Machines

Gregory B. Prokopski Clark Verbrugge

School of Computer Science
Sable Research Group
McGill University
Montreal, Canada

International Conference on Compiler Construction, 2008

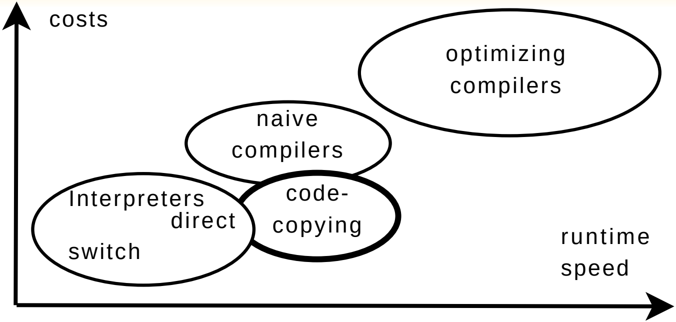
Taxonomy



Code-copying technique

Interpreter and also a JIT.

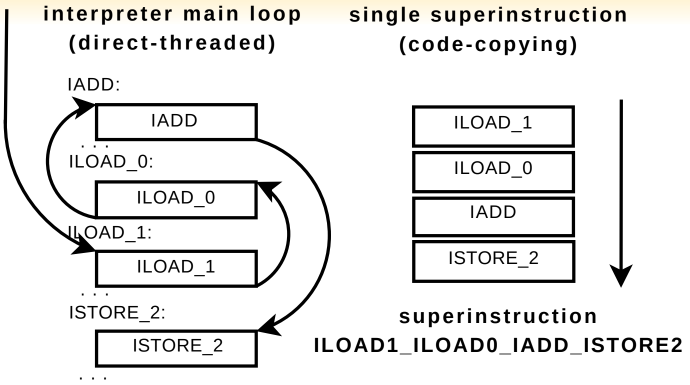
Speed Comparison



Code-copying technique

Bridges the performance gap while keeping costs low.
1.2–3.0 times faster than direct-threading.

Direct-threading vs. Code-copying



Code-copying technique

Reduces number of dispatches and improves branch prediction.

Code-copying Disaster Example

```
BCODE_START:
    if (...)
        {
            // then part
        }
    else
        {
            // else part
        }
```

How it happens?

- Direct-threading - one label

Code-copying Disaster Example

```
BCODE_START:
   if (...)
      {
          // then part
      }
   else
      {
          // else part
      }
BCODE_END:
```

How it happens?

- Direct-threading - one label
- Code-copying - two bracketing labels

Code-copying Disaster Example

```
BCODE_START:
   if (...)
      {
          // then part
      }
BCODE_END:
   . . .
      {
          // else part
      }
```

How it happens?

- Direct-threading - one label
- Code-copying - two bracketing labels
- Optimizations move basic blocks

Code-copying Disaster Example

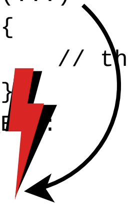
```
BCODE_START:
   if (...)
      {
          // then part
      }
BCODE_END:
   . . .

          // else ???
```

How it happens?

- Direct-threading - one label
- Code-copying - two bracketing labels
- Optimizations move basic blocks
- Incomplete copied code

Code-copying Disaster Example

```
BCODE_START:
   if (...)
      {
          // then part
      }
BCODE_E[illegible]:
   . .

          // else ???
```



How it happens?

- Direct-threading - one label
- Code-copying - two bracketing labels
- Optimizations move basic blocks
- Incomplete copied code
- CRASH!!!

Code-copying Disaster Example

```
BCODE_START:
   if (...)
      {
          // then part
      }
BCODE_END:
   ...

          // else ???
```

How it happens?

- Direct-threading - one label
- Code-copying - two bracketing labels
- Optimizations move basic blocks
- Incomplete copied code
- CRASH!!!

Problems always arise when a compiler uses relative addressing to reach outside a bytecode.

Motivation

Code-copying

- Easy, cheap to implement
- Great performance
- Not reliable (with modern compilers) - current approaches:
  - Ignore the problem.
  - Hand-check the assembly.
  - Trial and error testing.
  - Approximate runtime checks.

Outline

1. **Background and Motivation**
   - Interpreters vs. Compilers Gap
   - Code-copying and Its (Lack of) Safety
2. **Our Design and Implementation**
   - Copied Code Tracking
   - Verification
3. **Results and Conclusions**
   - Performance
   - Compiler Maintainability Impact

Copyable Code - What Is It?

Copyable Code "Chunk" Requirements

- Contiguous in memory between two labels
- Control flow "top" to "bottom"
- Jumps to outside and calls are absolute
- Jumps within chunk are relative
- Consistent registers use at entry and exit

Solution overview

Optimizing compiler (GCC) enhancement

- Programmer-friendly #pragma
- Track copyable code "chunks"
- Dozens of passes — Do not touch!
- Selective restore of code properties
- Final code verification

Solution overview

Inserted new passes

- Identify basic blocks of copyable code chunks
- Enforce absolute jumps and calls
- $\Rightarrow$ Run existing optimizations
- Basic block order fixup
- $\Rightarrow$ Legacy existing optimizations
- Copyable code verification

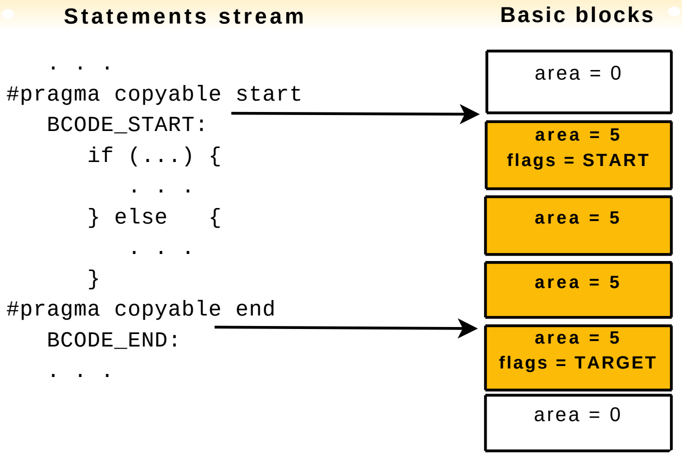
Pragma Handling

```
. . .

BCODE_START:
   if (...) {
      . . .
   } else   {
      . . .
   }

BCODE_END:
. . .
```

Pragma Handling

```
   . . .
#pragma copyable start
   BCODE_START:
      if (...) {
         . . .
      } else   {
         . . .
      }
#pragma copyable end
   BCODE_END:
   . . .
```

Pragma Handling



First and past-last basic blocks are marked as Start and Target.

Enforcing Absolute Gotos and Calls

```
BCODE_START:
   . . .
   if (ptr==NULL)
     goto NPE_handler;
   . . .
BCODE_END:
```

Need to correct relative addressing within "chunks".

- External assembler decides on addressing mode — not GCC.
- Needed an architecture-agnostic solution.

Enforcing Absolute Gotos and Calls

```
BCODE_START:
   . . .
   if (ptr==NULL)
     goto NPE_handler;
   . . .
BCODE_END:
```

```
{  void *target = &NPE_handler;
   __asm__ __volatile__ (
      ""             :
      "=r" (target) :
      "0"  (target) :
      "memory");
   goto *target;
}
```

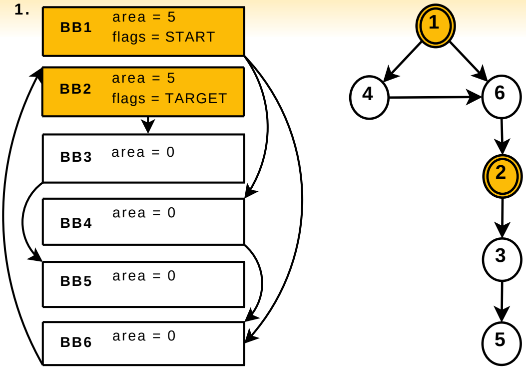
Need to correct relative addressing within ”chunks”.

- External assembler decides on addressing mode — not GCC.
- Needed an architecture-agnostic solution.
- Goto to the outside of chunk is forced into a computed goto.
- Each call is forced into call via function pointer.

Compiler Runs Largely Unaffected

- Once Start and Target basic blocks are marked and absolute addressing enforced all optimizations are performed as usual.
- A lot of work to modify several dozens of passes — don't!
- Start and Target block are never removed or duplicated.
- Able to find all copyable code of each chunk via CFG.
  - Traverse CFG from Start until Target or computed goto is reached.
  - No heuristics.

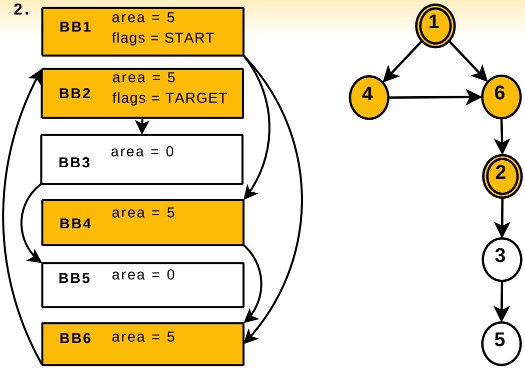
Ensuring Copyable Code Contiguity



1. Compiler moved basic blocks.

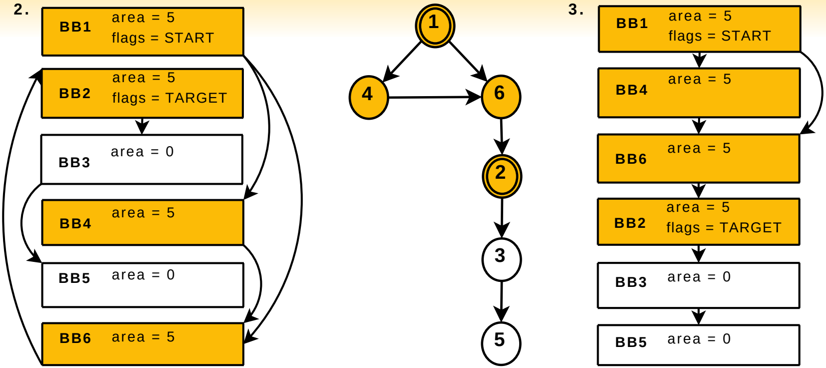
Ensuring Copyable Code Contiguity

2.



1. Compiler moved basic blocks.
2. Follow CFG to find blocks of each chunk.

Ensuring Copyable Code Contiguity



1. Compiler moved basic blocks.
2. Follow CFG to find blocks of each chunk.
3. Reorder basic blocks, deoptimize to ensure chunk contiguity.

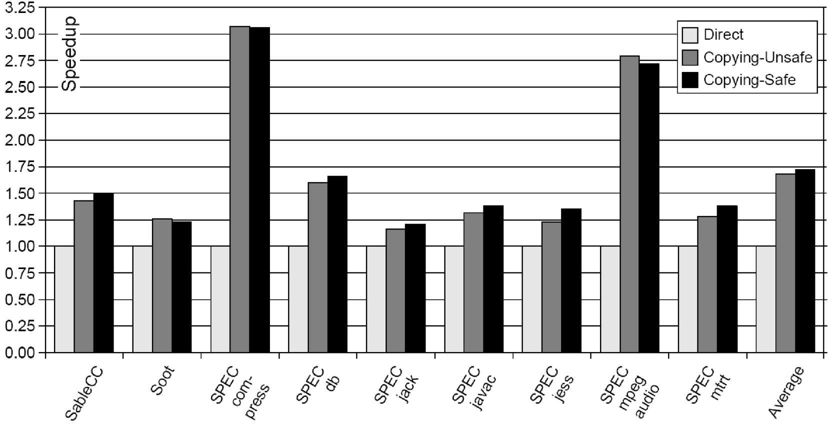
Final Verification Pass

- CFG is discarded at some point.
- Some legacy optimization code is ran after.
- Need to be sure of the final result.
- Insert special RTL "notes" to mark Start and Target.
- When the code is final verify all properties.
- This way ensure safety of the final result.

Brief Design Summary

- Enables **safe** code-copying.
- Avoided modifying dozens of passes.
- Very maintainable.
- Easy to use.
- Portable.

Performance Comparison



Comparable or faster than unsafe code-copying of SableVM JVM

Compiler Maintainability Impact

Metric	#
Data structures modified	4
Fields added to data structures	6
Data structures added	3
Functions added to existing files	4
Function calls/hooks inserted	8
Code lines added or modified	139
Code lines in new files	1500

- Minimal impact in terms of source modified.
- Update GCC 3.4 to 4.2 (2 years of development) took only a few hours.

Conclusions and Future Work

- Presented an industry compiler extension supporting copyable code generation.
- Easy to use by VM programmers.
- Easy to maintain in the compiler.
- Provides safety guarantees for copied code execution in a VM.
- Provides comparable performance to unsafe copied code execution.

- Expected future application to other VMs and other architectures.

Conclusions and Future Work

- Presented an industry compiler extension supporting copyable code generation.
- Easy to use by VM programmers.
- Easy to maintain in the compiler.
- Provides safety guarantees for copied code execution in a VM.
- Provides comparable performance to unsafe copied code execution.

- Expected future application to other VMs and other architectures.

Questions?